Naval Research Laboratory
Stennis Space Center, MS 39529-5004

NRL/FR/7441--95-9641

Object-Oriented Database Design and Implementation Issues for Object Vector Product Format

MARIA A. COBB
MIYI J. CHUNG
KEVIN B. SHAW

Mapping, Charting, and Geodesy Branch
Marine Geosciences Division

DAVID K. ARCTUR

University of Florida
Gainesville, FL

July 15, 1996

Approved for public release; distribution unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OBM No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	July 15, 1996	Final

4. TITLE AND SUBTITLE

Object-Oriented Database Design and Implementation Issues for Object Vector Product Format

5. FUNDING NUMBERS

Job Order No. 5745137A6
Program Element No. 0603704N
Project No. R1987
Task No.
Accession No. DN257086

6. AUTHOR(S)

Maria A. Cobb, Miyi J. Chung, Kevin B. Shaw, and David K. Arctur*

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
Marine Geosciences Division
Stennis Space Center, MS 39529-5004

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL/FR/7441--95-9641

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
Marine Geosciences Division
Stennis Space Center, MS 39529-5004

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

*University of Florida, Gainesville, FL

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Object Vector Product Format (OVPF) project undertaken by the Naval Research Laboratory and the University of Florida, and sponsored by the Defense Mapping Agency, successfully demonstrated the advantages that object-oriented (OO) technology can provide for importing and editing vector-formatted data. The final stage of this project, the design and implementation of an OO vector database supported by a commercial OO database management system (ODBMS) for the transformed vector data, is documented in this report. We begin with an overview of the OVPF framework and a discussion of ODBMS concepts. A comparison of two leading ODBMSs, GemStone and ObjectStore, is then presented with respect to issues such as architecture, security, migration policies, and use of transactions. This is followed by a discussion of specific issues related to the implementation of OO databases for OVPF using both GemStone and ObjectStore, including design decisions, required modifications to OVPF, and migration of data to the database. Finally, advantages and disadvantages of both systems realized from this experience are presented.

14. SUBJECT TERMS

MC&G data, mapping, database, object-oriented database, VPF, OVPF, ODBMS

15. NUMBER OF PAGES
20

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Same as report

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

CONTENTS

OBJECT-ORIENTED DATABASE DESIGN AND IMPLEMENTATION ISSUES FOR OBJECT VECTOR PRODUCT FORMAT

1.0 INTRODUCTION

This report documents the integration of the commercial object-oriented database management systems (ODBMS) GemStone (by GemStone Systems, Inc.) and ObjectStore (by Object Design, Inc.) with the Object Vector Product Format (OVPF) Smalltalk prototype, including both design and initial implementation. The work documented in this report was performed as part of the *Object-Oriented Database Exploitation within the Global Geospatial Information & Services (GGI&S) Data Warehouse* project conducted by the Naval Research Laboratory and the University of Florida and sponsored by the Defense Mapping Agency. The purpose of the project is to determine the potential impact of object-oriented (OO) technology on DMA's Global Geospatial Management Information and Services (GGMI&S) initiative. Other reports (Shaw 1995; Chung 1995a; Arctur 1995a, b) have documented the integration of multiple Vector Product Format (VPF) products into OVPF, network investigation results, and evaluation of a hybrid object-relational database management system (DBMS).

This document specifically addresses the design issues, as well as the experiences incurred, lessons learned, and potential impact from the actual implementation of GemStone and ObjectStore, two leading commercial ODBMSs, with respect to OVPF. Section 2.0 provides background information on object-oriented symbology and terminology, along with a brief description of the OVPF structure, including the design of the class hierarchies for importing and representing multiple VPF products. ODBMS concepts are presented in a general discussion in Sec. 3.0. Section 3.0 also includes an introduction to those OVPF design issues necessary for understanding the database implementation presented in Sec. 4.0. Section 5.0 concludes with a summary comparison of both databases' strengths and weaknesses.

The OVPF prototype application has been designed and implemented with the ParcPlace Systems' VisualWorks version of the Smalltalk programming language. This choice is primarily due to the sophistication and productivity of the Smalltalk development environment for building complex applications. Due to the semantic differences among OO programming languages, the terms and definitions used here may not apply consistently across all such languages. However, the concepts represented by these terms should be general enough to be implemented in other OO languages. The commercial ODBMS products have been chosen with this in mind as well—both ODBMS products include interfaces to Smalltalk and C++ so that objects created and stored by a Smalltalk program should be accessible to programs written in C++. This is not yet possible with current products, but industry efforts to develop cross-language compatibility for ODBMSs are underway.

2.0 OVPF DESIGN

2.1 Introducing Object-Oriented Class Diagrams and Terms

Figure 1 presents a partial cross-section of the class hierarchies designed to support multiple products. First, notice that some class names in Fig. 1 are underlined, while some are not. Classes whose names are underlined are called *abstract superclasses*, meaning that they represent a definitional abstraction (such as definition of instance variables and/or behavior to be shared by their respective subclasses) and that instances would not normally be created from them. Classes whose names are not underlined are called *concrete subclasses*, meaning that they are expected to have instances made from them. These terms are mainly used to aid in learning about a class hierarchy; to call a class "abstract" implies that it lacks behavior needed for creation of a "useful" instance-object.

Figure 2 presents a partial view of the data structures defined for each of the key feature definition hierarchies, while Fig. 3 presents a partial example of feature data and metadata objects that might be seen after importing Digital Nautical Chart Coastline (DNC COASTL) features. These *object diagrams* have two main sections: (1) the class name and (2) a list of instance

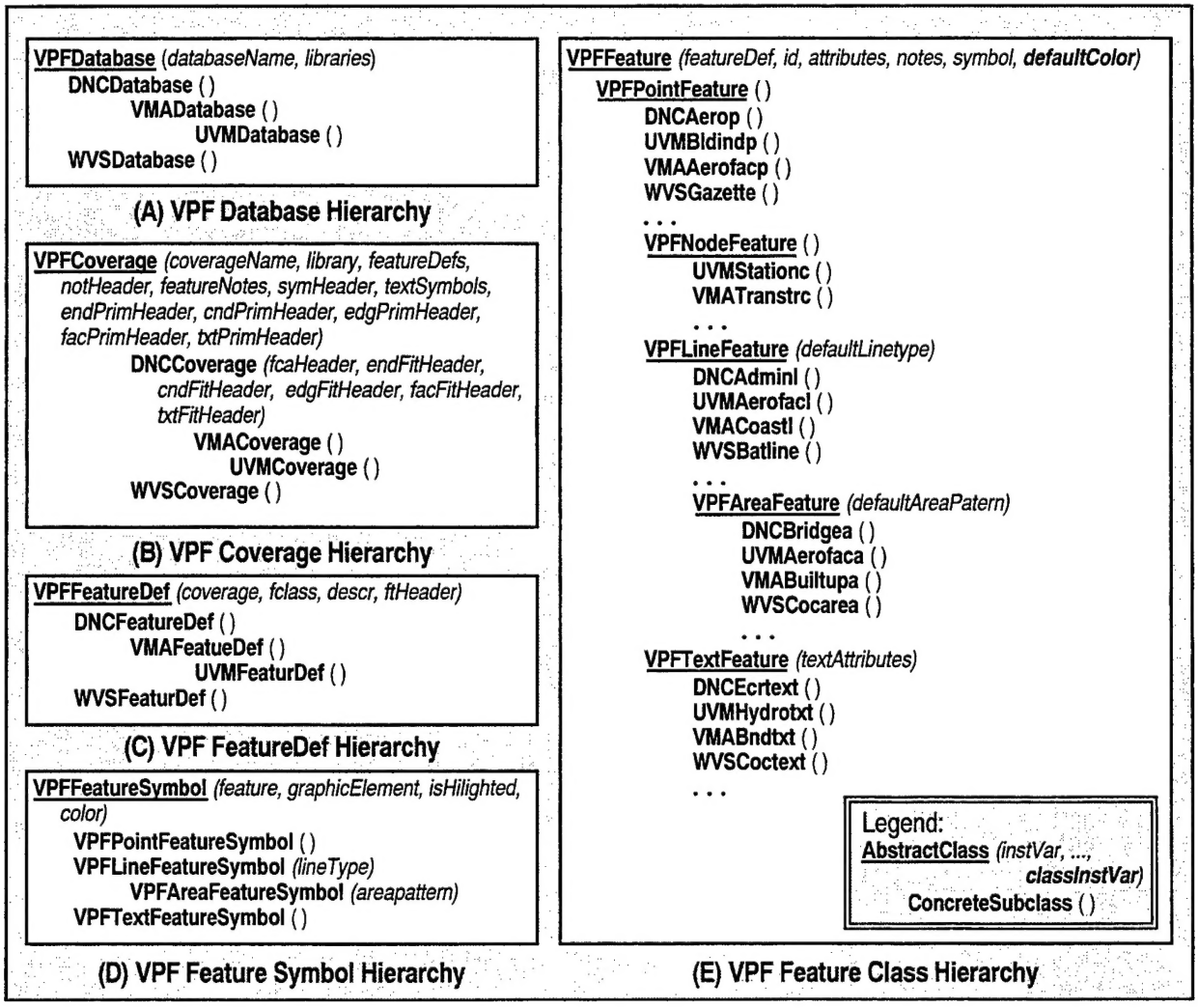

Fig. 1 — Selected OVPF class hierarchies with instance and class-instance variables indicated

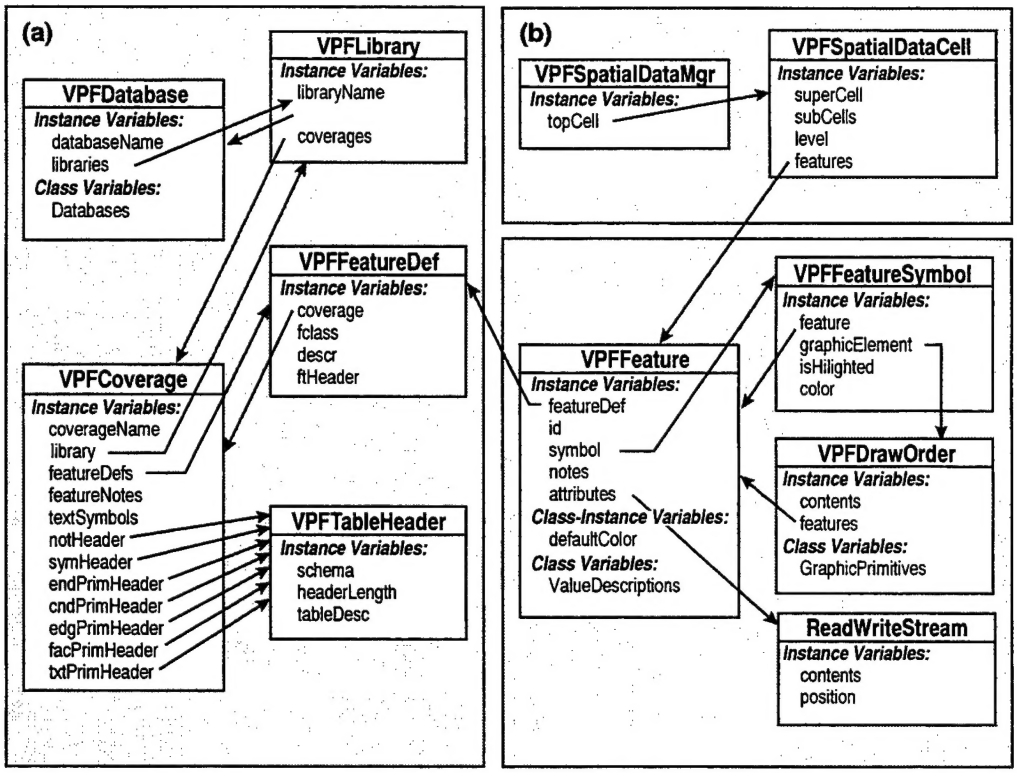

Fig. 2 — Key definitions and relationships for OVPF database classes, (a) metadata and (b) feature data

variables, class-instance variables, and/or class variables (where applicable). These variables will each be described briefly in turn.

Instance variables are data structures for which each instance-object has its own private copy, such as a feature's *id*, *attributes*, and *symbol*. Instance variable definitions in one class are inherited as part of the definition by all of its subclasses. For example, instances of DNCAerop and all other VPFFeature subclasses will inherit the definition of a private copy of *id*, *symbol*, *notes*, and other instance variables defined in their superclasses. According to Smalltalk convention, instance variable names begin with a lowercase letter.

Class-instance variables are data structures for which *the class object and each of its subclasses* are defined to each have a private copy of the variable. A class-instance variable can be used, for example, to hold a subclass-specific default value for a constant that can be accessed with the same name from any class in the hierarchy. This helps reduce the program's "variable-name vocabulary," which is one of the benefits of OO design. Class-instance variable names are shown ***bold-italicized*** in Fig. 1 to help distinguish them from *instance variables*.

Class variables are data structures for which the defining class has a single copy that can be directly accessed by all instances of itself and its subclasses. Per Smalltalk convention, class variables (and other shared objects including classes) have names that begin with an uppercase letter. Class variables are generally used either to hold (1) application-specific constants or (2) collections of specific instances of a class. In Fig. 2, examples of class variables include: (1) *Databases* (in VPFDatabase class) used to hold pointers to all the OVPF product metadata object webs,

(2) *GraphicPrimitives* (in VPFDrawOrder class) used to hold all instances of VPFDrawOrder, and (3) *ValueDescriptions* (in VPFFeature class) used to hold all the INT.VDT and CHAR.VDT file contents.

Note the use of *collection-objects* in Fig. 3—these are instances of predefined classes in Smalltalk that are used to encapsulate a collection of one or more types of other objects. Collection objects simplify the management of large groups of objects when it is important to be able to sequentially or randomly access the group members. Libraries, coverages, featureDefs, and feature objects are frequently held and accessed from collection objects in OVPF. In contrast, VPFSpatialDataCells do not need a collection object because they are used as a linked list. Also, each VPFFeatureSymbol is uniquely bound to one VPFFeature object, so it does not require a separate container.

2.2 Metadata Classes and Instances

It will be helpful now to introduce the metadata (schema definition) classes in OVPF. VPFFeatureDef and its subclasses form an important link between feature objects and their defining schemas. Figures 1, 2(a), and 3(a) show the classes that define the structure of each VPF database product. The classes VPFDatabase, VPFCoverage, and VPFFeatureDef all have product-specific subclasses. The need for product-specific subclasses is most concerned with the feature import process and will be discussed in Sec. 2.3. What is more pertinent for now is that much of the feature

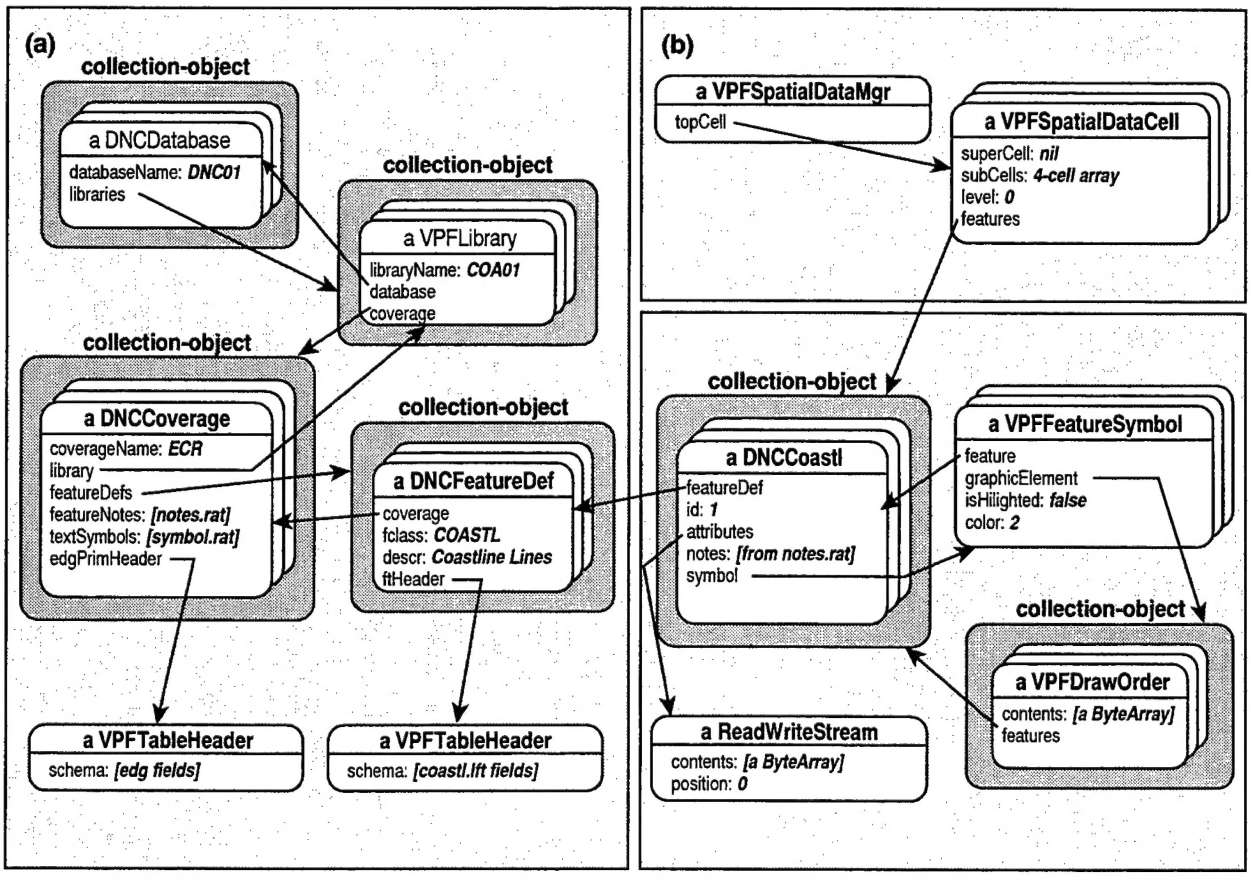

Fig. 3 — Sample of OVPF data for a DNC coastline feature, (a) metadata objects and (b) feature data objects

metadata is stored in instance variables whose names end in "Header," such as *ftHeader* in VPFFeatureDef class. These variables each hold onto an instance of VPFTableHeader, which in turn holds a schema of information such as the field id, description, data type, and length of each column in the table, as documented in the VPF product specification. Another role of the VPFTableHeader schema is to support the actual conversion of source data bytes into usable numeric values and vice-versa, as needed.

2.2.1 Feature Attributes

As shown in Fig. 1, each subclass of VPFFeature inherits an *attributes* instance variable. The *attributes* variable holds onto the contents of a record from the *.PFT, *.LFT, *.AFT, or *.TFT feature table. Each of these records contains the integer or character values of attributes common to all features of a given class, such as ACC (accuracy code) or SLT (shoreline type).

Figure 1(e) shows that VPFTextFeature also defines a *textAttributes* instance variable. This variable holds onto a set of values from the SYMBOL.RAT file (if present) that defines text font, style (italic, bold, etc.), size, and color for one or more text features.

Both *attributes* and *textAttributes* are stored and processed in the same manner. They use a predefined Smalltalk class called ReadWriteStream that provides an efficient means of storing, reading, and updating an arbitrary sequence of bytes. Since floating-point and integer values are stored somewhat differently among the various computer systems (UNIX, DOS, and Macintosh), we chose to defer translating the bytes into user-understandable values until the user actually requests to examine or modify them. In addition to reducing processing time during feature import, this also greatly reduces the total number of objects we have to handle. Since we are using a single ReadWriteStream object for each set of attributes instead of creating a separate integer or character value object for every attribute, we are greatly reducing the load on the computer system's internal I/O channel and memory.

Since these attributes are normally maintained as an arbitrary sequence of bytes within OVPF, it is necessary for a given feature object to have access to the byte-offset and length of each of its attributes in the ReadWriteStream object. This information is read and stored during metadata initialization of a VPF database, and is held in the *ftHeader* schema defined in VPFFeatureDef (for attributes defined in feature table headers) and in the *symHeader* schema defined in VPFCoverage (for text attributes defined in SYMBOL.RAT).

2.2.2 Integer and Character Value Description Tables

One set of tables all VPF products have in common includes the *integer value description table* (INT.VDT) and the *character value description table* (CHAR.VDT). Each coverage has these VDT files to store the descriptive text associated with each of the valid ranges of feature attribute values. In OVPF, the set of all VDT data is held in a class variable of VPFFeature class called *ValueDescriptions*. This is organized as a nested hierarchy of tables, so that the valid range of integer and textual values for a given attribute is easily found. The *ValueDescriptions* structure is populated during database initialization.

Holding all of these VDTs in a single hierarchical collection structure has proven useful in identifying inconsistencies among values and descriptions across coverages, libraries, and databases. Some of these differences reflect errors in the data, but most are due to design differences in the use of a given feature attribute among different VPF products.

2.2.3 Feature-Related Notes

If a coverage includes a NOTES.RAT file, this is read into the VPFCoverage subclass' *featureNotes* instance variable during database initialization. A NOTES.RAT file normally has many-to-many links with feature objects; that is, any one feature may be linked to many records in the NOTES.RAT file, and any one note in the NOTES.RAT file could be linked to many different feature objects. The purpose of the *notes* instance variable defined in VPFFeature is to hold onto a list of note IDs from the NOTES.RAT file for lookup into the VPFCoverage subclass' *featureNotes* collection. The *notHeader* schema defined in VPFCoverage is used when reading these notes from disk. So far, DNC appears to be the only VPF product using NOTES.RAT.

2.2.4 Graphical Primitives

Each feature object is associated with a set of latitude-longitude coordinates, referred to as graphical primitives. Point and node features are associated with entity- and connected-node primitives. Line features are associated with edge primitives. Area features are associated with a face primitive consisting of a "ring" of edge primitives, and text features are associated with a "path" of coordinates. Because any one-line feature object may consist of multiple edges, and any single node, edge, or face primitive could be used by more than one feature object, great care must be taken to maintain the correct linkage between the features and primitives. Topological relationships (adjacency and contiguity) among the primitives must also be maintained across all features within a given coverage and tile, according to the VPF specification.

The design of spatial topology within OVPF is documented in Chung (1995b) and it is not an intention in this report to elaborate on the intricacies of such. Therefore, for purposes of this report, graphical primitives will be treated as if all were instances of the same class and will be called VPFDrawOrder. This approach is sufficient for explaining the manner of encoding each primitive's location coordinates. Also, VPFDrawOrder is the superclass from which all the topological primitive classes are derived in the topology framework.

As seen in Figs. 2(b) and 3(b), VPFDrawOrder objects have a pointer to a collection of feature objects. This is to allow each graphical primitive (draw order) to know and communicate directly with the feature(s) in which it is used. Figure 2(b) also shows a class variable *GraphicalPrimitives* defined for VPFDrawOrders. This is to hold the set of all VPFDrawOrder instances imported or created in the current work session.

VPFDrawOrder objects hold all location coordinates in a specially encoded array of bytes and work much like the ReadWriteStream used to store and process a feature's attributes. A VPFDrawOrder object has an instance variable *contents* that contains the byte array. This byte array has three parts:

- opcode byte–an 8-bit integer used to define what kind of operation is to be performed; e.g., 199 means "set polyline"
- length byte–an 8-bit integer containing the number of data bytes to follow
- data bytes–sequence of bytes representing a color index, location coordinates, etc.

Instances of VPFDrawOrder are used to represent nodes, edges, the polylines forming each character in a text feature's string, and any other graphical entity associated with a feature object. These VPFDrawOrders can be concatenated into arbitrarily long sequences of bytes for any given purpose. This is a computationally efficient means of managing the storage and processing of the tens and hundreds of thousands of coordinate points that are required to represent a VPF database.

2.2.5 Spatial Tree Indexing Framework

Two classes shown in Figs. 2 and 3, VPFSpatialDataManager and VPFSpatialDataCell, are used to implement and manage a spatial tree indexing framework. This framework presently uses a quadtree organization in which each spatial data cell holds pointers to four subcells and each subcell holds a pointer to its parent or supercell.

This spatial tree design is independent of the VPF database to be imported. All access to the spatial tree from within OVPF is done through a spatial data manager object. Each feature passed to the spatial data manager is inserted into the appropriate spatial tree cell based on the feature's minimum bounding rectangle. The spatial data manager also responds to user requests to obtain or delete the features appearing within a particular region in space.

This design allows us to modify the implementation of the spatial tree at any time without affecting the rest of OVPF or the source data. Thus, in the future, we could easily substitute an R^+ tree (Stonebraker 1986), PM Random (PMR) quadtree (Nelson 1987), spatial splay tree (Cobb 1995), or special optimizing techniques in place of the present quadtree approach. We could also support the simultaneous implementation of multiple spatial indexing schemes to allow choice of the most efficient spatial tree design for a given source database. This may be important as we provide support for Raster Product Format (RPF) and other non-VPF databases.

3.0 ODBMS INTEGRATION

3.1 Why Use an ODBMS?

The function of any DBMS is to provide persistent (maintained from session to session) storage of data, controlled access to the data, and backup and recovery capabilities, among others. Object-oriented DBMSs provide these functions specifically for *objects*—units of data defined and assigned values through the use of an OO programming language such as Smalltalk or C++. While objects are generally considered to consist of both state (data) and behavior (procedures), ODBMSs are typically concerned only with the storage of the state information, as are traditional relational database management systems (RDBMS).

With support for multiple VPF products now integrated into OVPF's framework, we turn to the issues and tradeoffs of support for commercial ODBMSs. This section presents our current experience and understanding with respect to the issues and effects of alternative ODBMS architectures on OVPF's design. In this regard, we will be distinguishing between internal and external databases. The internal database refers to OVPF's computer-memory-resident object space of metadata and feature objects. The external database refers to the disk-resident database implemented using the commercial vendors' ODBMS products as an alternative to the relational VPF file structure.

In today's distributed processing environment, each OVPF user is likely to be working on a networked computer workstation that is separate from, but perhaps just as powerful as, the workstation housing the shared external databases. OVPF will be seen as a *client* process making requests for data from a *server* process running on the central host. Commercially available ODBMSs typically fall into one or both of two types of client-server architecture: object-server or page-server. The distinction is based on the unit of data transfer between the server and client processes. In an *object-server* architecture, the server understands the client's concept of an object. In this model, each transfer from the server to a client process is based on groupings of interconnected

objects. A *page server,* on the other hand, is unaware of "objects" as such, but transfers data in units of a disk page that is typically 4 Kb.

As described above, OVPF can function without a database management system. Relational tables as stored in directories according to the VPF specification (Defense Mapping Agency 1993) are processed and information brought into memory upon import of one or more coverages. Once in memory, disk resident data is never subsequently accessed. The advantage to this approach, besides its simplicity, is that the memory resident data is quickly accessible for manipulation, eliminating the need to perform costly disk accesses and table joins. The disadvantages are primarily: (1) the amount of data that can be imported for a single session is limited by the capacity of physical memory and (2) data is not made available for concurrent access by multiple users; thus, changes to the data made through the use of OVPF are not readily apparent to others.

The use of an ODBMS eliminates both of these concerns, as well as providing additional advantages. For example, with this approach, OVPF is no longer limited by memory size for data import and viewing; data is simply stored in the database until needed, then brought into memory for display or editing purposes. Additionally, geographic object level security and auditing can be readily managed.

3.2 ODBMS Concepts

Following is a list of three significant high-level concepts concerning ODBMSs. Each is elaborated upon in the discussion that follows.

- persistent vs. transient objects
- transactions and concurrency control
- security and authorization

The distinction between persistent and transient objects is somewhat less clear for ODBMSs than for RDBMSs, due to the tightly coupled nature of the database with the application. *Transient objects* are defined to be those objects that exist in the computer's memory only during execution of the application, such as the window system objects for displaying the key data. *Persistent objects*, on the other hand, are those representing the key data, whose state is maintained in the database and exist even when there is no application program running.

Transient and persistent objects can coexist within a running process. The distinction becomes blurred because persistent data accessed from the database can be assigned to a transient object. It is important for application programs to manage both transient and persistent data in consistent ways so updates to persistent data are made when needed and so data that should remain transient are not inadvertently made persistent. For example, a web of interconnected transient objects can be made persistent simply by allocating space in the database for the "root object" of the web. Once this transaction is completed, the ODBMS will migrate the transitive closure (entire web) from the root object into the database. It is important, therefore, to be sure that such a transitive closure does *not* include references to objects, such as the window system in which the data is being displayed, as this would "pull" a very large and unnecessary part of the application objects into the database.

All accesses to persistent data are typically made within a *transaction*. A transaction is defined to be a sequence of instructions that access the database and whose execution is guaranteed to be atomic; i.e., either all or none of the instructions are executed. *Read* transactions are defined as those that retrieve data only, while *write* (or *read-write*) transactions are those that actually change the data. When a transaction has successfully terminated, it is *committed*, meaning that any changes

made to persistent data within the transaction are written to the database and will not be undone. If a transaction is *aborted*, the ODBMS will cause a rollback of changes made to the objects to return them to their state as existed prior to the beginning of the transaction.

A related issue is that of *concurrency control*. Concurrency control is an issue only for multiuser DBMSs and is concerned with ensuring, usually through the use of locks, that when a user is accessing persistent data within a transaction, other users cannot simultaneously change the data, resulting in the database being left in an inconsistent state. Database inconsistency can easily result from the interleaving of different users' instructions regarding the same persistent data.

It is widely recognized that data is one of the most valuable assets of any organization. With that recognition comes the need to protect such data. As a result, DBMSs typically provide some system of restricting access to data based on a user authorization system. Different sets of privileges may be provided for different classes of users. Privileges may include the right to read or modify data, as well as the ability to modify the database schema itself.

3.3 ObjectStore vs. GemStone

ObjectStore and GemStone were selected for integration with OVPF for the following reasons:

- both are well-established, commercially successful ODBMSs
- each is available with both Smalltalk and C++ language interfaces
- they are representative of two different client-server architectures

A brief explanation of several of the most significant differences in the two database management systems will be given below; however, for a more detailed explanation, the reader is referred to Shaw (1995).

3.3.1 Architecture

The most distinguishing feature of the ODBMSs is the server architecture. GemStone is based on an *object-server* model, while ObjectStore is based on a *page-server* model. In this context, a *page* refers to a disk page, typically 4 or 8 Kb. Both types of servers transfer data from the database to the executing program in units of disk pages. The main distinction between the servers is that the object server has knowledge of the logical organization of the database and can perform certain operations, such as user authorization and query filtering, at the server *prior* to transferring any data to the client. The page server, on the other hand, has no knowledge of the logical organization, but relies on the client process to manage all authorizations and query optimization. A previous study (DeWitt 1992) found that the page-server approach was better than a pure object-server approach in terms of raw data throughput performance. However, GemStone has incorporated a page-server foundation into its object server, reducing the potential advantage ObjectStore might have in performance alone.

An important aspect of these server models is the consequence for maintaining application code and data integrity while multiple users and/or programmers are working on the system. In GemStone's object-server model, local replicates of persistent objects in a user's application memory can become out of date with the central database; this is referred to as the "two-space" problem. Additional overhead in program design and client-server interaction is required to address this issue. ObjectStore

does not allow local replicates of persistent data to be held in the client application, essentially eliminating the two-space problem.

3.3.2 Migration Policy

The manner in which an ODBMS handles migration of transient objects to persistent objects upon transaction commit is known as a *migration policy*. This policy defines conditions upon which a given transient object will be migrated to the database. Both ObjectStore and GemStone allow for migration policies to be defined at the class, instance, or instance variable levels. The default policy is to make the transitive closure of a migrated transient object persistent; however, this can be overridden to prevent specific components or links of a migrated object from being made persistent.

3.3.3 Transactions

ObjectStore and GemStone differ in their philosophy for transactions, possibly as an outcome of the difference in server models. The first issue is the *requisite* use of transactions for accessing persistent data. ObjectStore uses a "pessimistic" approach that requires every database access to occur within the bounds of a transaction. This guarantees data integrity among all users at the cost of potentially blocking some users while one is accessing a particular page of data. GemStone, however, requires only those statements that may change persistent data to be encapsulated within a transaction, i.e., applications may read and even lock data outside of a transaction. This "optimistic" approach is more efficient for users who are interested only in examining the data without modifying it and by not incurring the overhead costs associated with transactions. The disadvantage is that users may read "dirty" data—data that has been read and possibly changed by another user. This approach can result in the need to abort the second user's transaction, while the pessimistic approach would have prevented the second user from ever having viewed the data that was accessed within the first user's transaction. The relative advantages of one approach over the other depend primarily on database usage patterns.

Another difference in the way in which GemStone and ObjectStore handle transactions is in the subject of *implicit* vs. *explicit* transactions. ObjectStore requires users to explicitly state the bounds of all transactions by the use of messages to ObjectStore's class OSTransaction. This helps ensure that each transaction will be as short as possible, thus minimizing blocking of other users. GemStone, however, has an "automatic transaction" mode in which a new transaction is started automatically whenever a user logs in and after every commit or abort. This mode is actually very helpful during application development and can work well in applications that have low concurrency and/or low transaction commit rates. The user may override this facility if desired and explicitly state the beginning and ending of transactions as in ObjectStore.

3.3.4 Security

ObjectStore at present has no explicit capability for administering user- and group-level access permissions to the data, although it has been announced for their 1996 product release. In contrast, GemStone has a framework that allows varying levels of protection. The first is the control of log in authorization through a user id and password. Another is the organization of groups of objects into *segments*, a structure that is owned by a single user and that has permission settings for the owner, several groups, and the "world." Thus, permission for reading or modifying certain data could be assigned at a group level for multiple users at a time.

4.0 DATABASE IMPLEMENTATION

The implementation of each database necessitated the following steps:

1. Making design decisions such as:
   - deciding which objects should be transient and which objects should be persistent
   - deciding on physical partitions of the database
   - deciding on roots of persistent object trees
   - establishing a security model.
2. Modifying the existing implementation of OVPF in the following ways:
   - reorganizing or creating new classes and methods needed for database use
   - inserting correct references for persistent objects and ensuring that no persistent object references are permanently held in the application program
   - placing transaction boundaries in appropriate places.
3. Loading the database with the VPF data, including:
   - building the initial database file
   - importing and migrating the metadata
   - importing and migrating the feature data.

The relation of each of these steps to both GemStone and ObjectStore is discussed in detail in the following section.

4.1 Persistent Object Webs in OVPF

The first step of the external database design involves the decision regarding which sets of objects to make persistent. This is very different for OO than for relational DBMSs. Any object within OVPF's internal memory can become a persistent object in an ODBMS merely by "asking" it to be. However, this not only copies the requested object into the external database, it also copies the object's transitive closure of every object to which it points. However, it is often undesirable to store some objects in such a transitive closure in the external database. Each of the ODBMS products provides a means to bypass the transitive closure under certain conditions so that only selected links are followed. The implications of this issue within OVPF are discussed below.

In this report, three main groupings of database objects have been presented: the metadata object web, the feature objects, and the *ValueDescriptions* data structures. Each of these object groups is made persistent. Another category of OVPF objects includes the user interface classes. These are the support classes that present the map on the computer screen and allow interaction with the user. Instances of these classes should not be made persistent.

Typically, a complete web of objects is made persistent by reference to some "root" or "parent" object for the group. This also provides a named entry point to the persistent object web for future access by other application programs. In the case of the metadata object web, the root is a collection object containing pointers to all initialized databases, as shown in Fig. 3(a). For example, this collection has a member called DNC01 that points to its libraries, each of which points to its

respective coverages, and so on. In the non-database version of OVPF, this root collection of databases is held by the VPFDatabase class variable called *Databases*. However, with the integrated database, this metadata object web is made persistent, thereby alleviating the need for the *Databases* class variable.

For the feature objects, the logical root object is the spatial tree manager that holds a pointer to the linked list of spatial tree cells, each of which holds pointers to the features whose bounding rectangle falls within the cell's boundaries. Each feature object (instance of a VPFFeature subclass) holds onto its attributes stream and its symbol (instance of a VPFSymbol subclass).

For the graphic primitives, the logical root object is the collection of all VPFDrawOrders, presently held by the VPFDrawOrders class variable *GraphicPrimitives*. This root collection object is made persistent in the object database. Therefore, as was the case for the *Databases* class variable, there is no longer a need for the *GraphicPrimitives* class variable. Likewise, the *ValueDescriptions* structure held by the class variable of VPFFeature is handled in a similar manner.

4.2 ObjectStore

ObjectStore provides a relatively small set of low-level operations for database accessing and manipulation. Thus, integrating ObjectStore with OVPF was a fairly nonintrusive operation, requiring some restructuring of the OVPF class hierarchy, but few changes in fundamental design or implementation.

4.2.1 Conceptual Database Organization

One of the first changes made was a reorganization of the OVPF class hierarchies to change the usage of class variables. ObjectStore does not allow objects held by these kinds of variables to be made persistent; instead, all persistent state should be held by instance variables. For example, as discussed earlier, OVPF previously used class variables of the graphical primitive classes to hold centralized collections of all primitives organized by coverage. This collection was moved and divided to be held by instance variables of each coverage object so that each coverage now holds a collection of just its own primitives. Another class variable previously held the root of a tree structure of all value description table (VDT) entries in a given database. This collection was also moved and divided so that each feature's metadata (called its featureDef) object now holds a collection of just the VDT entries for that feature class. These and other similar changes took several days of programming to complete, and proved necessary for integration with GemStone as well.

4.2.2 Physical Database Design

ObjectStore facilitates clustering of data through the use of *segments*. A segment is a variable-sized region of storage composed of disk pages. Segments or pages can be specified as the unit of transfer of data from disk to memory, and can range in size from the default of 4 Kb to over 128 Mb. Unless otherwise directed, all data are automatically stored in a predefined segment called the *default segment*. However, it is usually desirable to create multiple segments, as the appropriate designation of segments for specific groups of objects can dramatically improve database performance by defining which objects are transferred together.

OVPF uses four segments, known as the *default segment*, the *spatial segment*, the *feature segment*, and the *primitive segment*. Each segment stores a group of related objects as given below:

- **default segment**–the database metadata
- **spatial segment**–data relating to the spatial index (quadtree)
- **feature segment**–all feature-level data
- **primitive segment**–all graphic primitive data; e.g., edge, face, node data.

This physical partitioning of the data was performed to support an optimal clustering strategy. This helps prevent unnecessary data from being transferred upon access to the database. Very little change was required in OVPF to define and access these segments.

ObjectStore allows multiple entry points to each database to provide efficient access to various logical hierarchies of data. Each entry point is referred to as a *named root* of persistent data. Each object designated as a root object can be accessed directly through the use of its name. All objects in the root object's composition hierarchy can then be accessed by navigating through its instance variables.

Two root objects were created for the OVPF database. These are OSDatabasesRoot, which provides access to the different VPF databases such as WVS+, DNC, etc., and OSSpatialIndexesRoot, which provides an entry point for the quadtree spatial indexing structure. In OVPF, the spatial indexing organization was modified somewhat so that each coverage has a pointer to its own quadtree in the database. Direct access to each quadtree is essential for fast responses to user queries, and the OSSpatialIndexesRoot provides access to the roots of all quadtrees in the current database. While the term *quadtree* is used in this report, OVPF is organized to support any combination of different types of spatial indices among VPF products, with each feature coverage specifying its own spatial index. Figure 4 shows the four OVPF database segments and their connections to each other. Database roots are noted through the use of the symbol®.

4.2.3 Proxies and OSReferences

The class OSReference is provided by ObjectStore to handle inter-object references from transient to persistent objects. An instance of the OSReference class replaces a reference from a transient to a persistent object upon transaction commit. OSReferences provide object identifiers that are valid across transactions. OSReferences forward messages within a transaction to the persistent objects they represent, and as such can be used in a manner similar to that of the persistent objects themselves. OSReferences were used in OVPF for instances of the metadata classes VPFDatabase, VPFLibrary, VPFCoverage, and VPFFeatureDef.

Proxies, implemented by the lightweight OSProxy class, are used when a handle to a persistent object is needed, but when references to that object do not necessarily require all of the object's data. A string can be assigned to a proxy object such that references to the persistent object represented by the proxy that occur outside of transactions will return the proxy's string. Any references made to the proxy while inside a transaction will return the persistent object. Figure 5 illustrates the use of proxies with OVPF.

4.2.4 Loading the Database

Once the database file has been created through use of the appropriate ObjectStore method, it may be loaded with the VPF data. This is done in two steps. First, the metadata is imported from

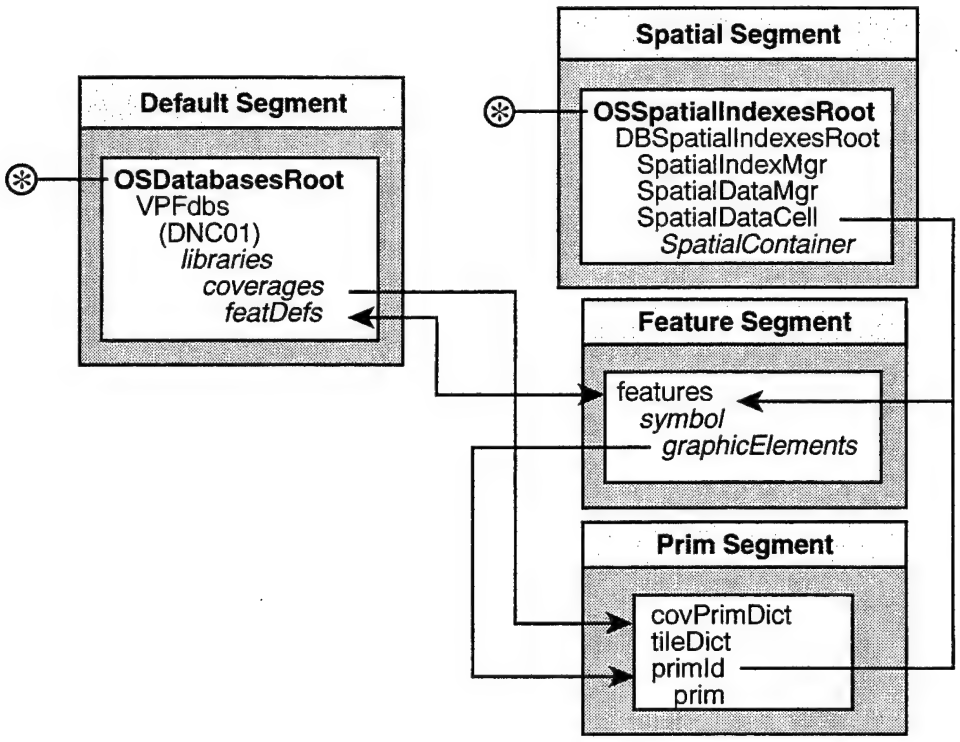

Fig. 4 — ObjectStore database segments and roots

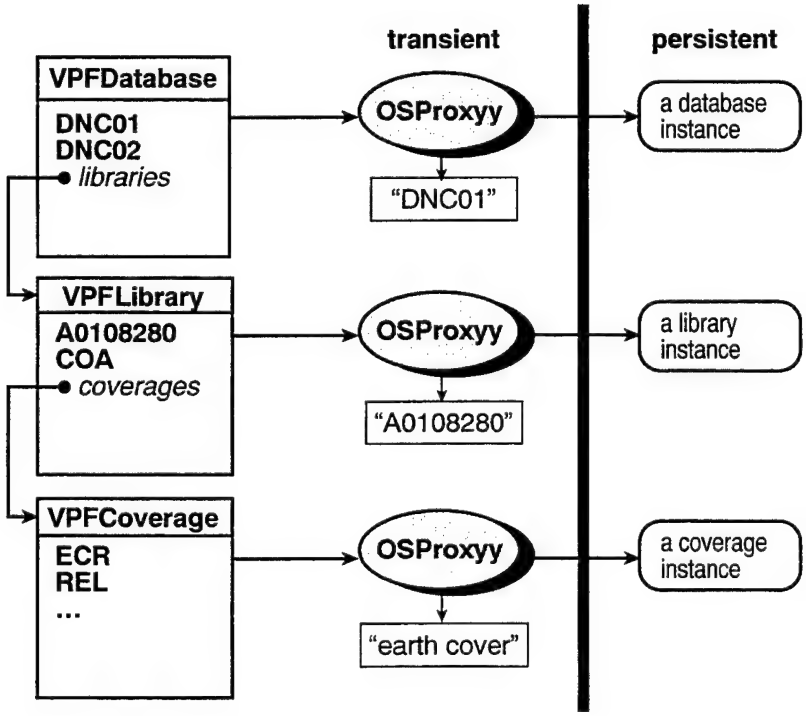

Fig. 5 — Use of proxies with ObjectStore

the VPF tables (metadata includes VPF tables such as the Database Header Table (DHT), Library Attribute Table (LAT), and Library Header Table (LHT), each feature class' schema, and the VDTs). After the metadata have been loaded and migrated to the database, feature data are imported from the VPF database, one coverage at a time, and placed in the memory-based quadtree spatial indexing structure. The features and the quadtree are then migrated to the database in their respective segments upon commit of the feature loading transaction. After the bulk loading of the databases in this manner, the relational tables are not used for subsequent data access—all manipulation of the data involves only the object database.

4.3 GemStone

GemStone is a somewhat more mature ODBMS, having been available with the Smalltalk interface for approximately 10 yr. Its client-server architecture is also more sophisticated, having provisions for application-level processing at both the client (Gem) side and the server (Stone) side. Server-side processing could be utilized to filter query objects, resulting in a reduction of network traffic necessary to satisfy users' data needs.

Figure 6 shows a high-level diagram of the GemStone architecture as integrated with OVPF. This diagram shows OVPF communicating to the Gem client processes (there will be one Gem process per VPF database) through the GemStone Smalltalk Interface (GSI). The Stone server processes requests from the Gems, retrieving objects from the GemStone database, and possibly performing processing on the resultant objects before passing them to the Gems. The data are then brought into the Smalltalk image.

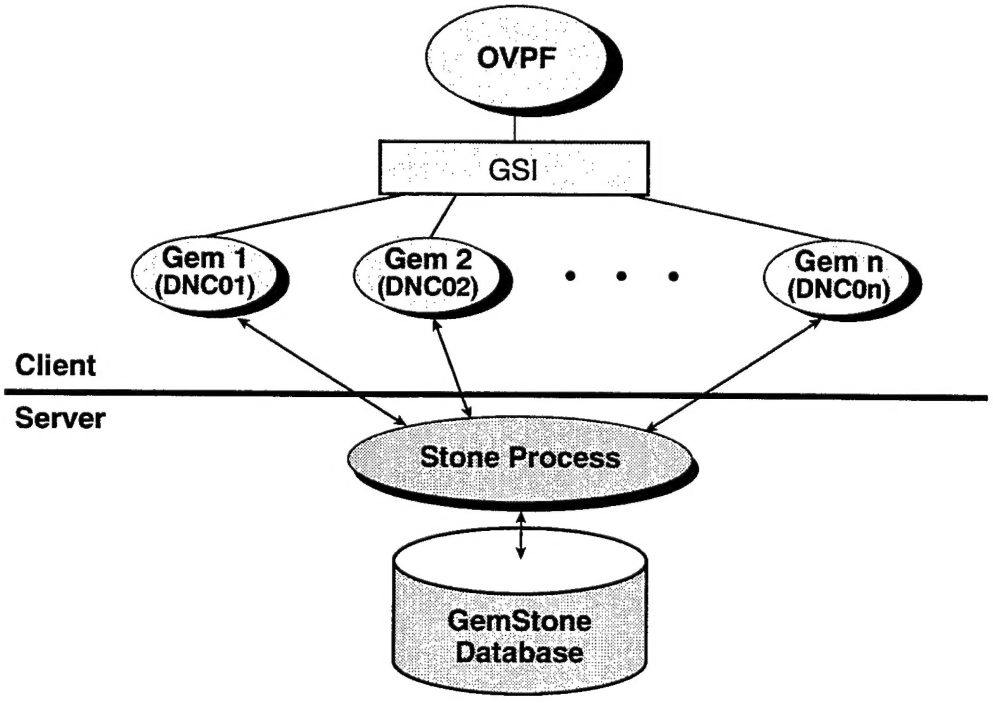

Fig. 6 — Overview of GemStone architecture

4.3.1 Design

Many of the design issues that were resolved for ObjectStore are also applicable for GemStone. For example, the decisions regarding which objects to make persistent, which objects should be database root objects, and where transaction boundaries should be placed are all valid for both ODBMSs. One difference regarding physical database design is that GemStone has no provisions for implementing user-defined physical partitioning of objects, such as that provided through the use of ObjectStore segments. Therefore, clustering strategies were not considered for GemStone. (*Although GemStone does provide structures known as segments, these segments are purely logical groupings of data that define the objects' accessibility to various groups of users, rather than physical disk partitions.*)

4.3.2 Stubbing

A technique known as *stubbing* is used in GemStone to control the number of levels of objects that are brought into memory from the database. Normally, whenever a persistent object is accessed, that object, along with the transitive closure of all objects accessible by it, are faulted into memory. However, many times this is unnecessary and can be very costly in terms of memory usage. A stubbing policy associated with each class limits the level of objects actually brought in when a member of that class is accessed in the database. Any object beyond the specified level is replaced by a *stub*, an empty placeholder that knows which GemStone object it represents. Whenever a stub receives a message, the stub is replaced by a replicate of the GemStone object it represents. Stubbing is useful for object collections with high fanout or deep hierarchies.

4.3.3 Transactions

As mentioned earlier, GemStone provides both automatic and manual transaction modes. Unless the application specifies manual mode, a transaction is automatically started upon login and again after every commit or abort. For the initial implementation with OVPF, automatic transaction mode was used simply because it reduced implementation time. This means essentially that the entire OVPF application runs within transaction boundaries. Normally, this is not considered desirable because running within a transaction implies locking of data and lower permissible levels of concurrency. However, given the nature of the use of OVPF, which is basically a low level of concurrent access to data together with a low commit rate (frequency with which the database is updated), long transactions such as those that result from this design are not extremely problematic.

Difficulties could arise with this use of lengthy transactions. In particular, when a user commits or aborts a transaction or when a lock is requested, that user's view of the database is refreshed before the action occurs. The problem with this is that during long transactions, much work could be lost if, for example, the user had used a particular database value for some calculations, then upon committing finds that the value had already been changed by another user. Having shorter transactions decreases the chances that this type of situation will occur. Future optimization plans include the incorporation of manual transactions. Transaction boundaries would then be equivalent to those used in ObjectStore.

4.3.4 Remaining Issues

Many of the sophisticated facilities provided by GemStone have not yet been fully utilized. Specifically, these include implementing groups and security administration models and symbol

lists for managing visibility of data. Three user groups are anticipated: a *metadata admin* group, a *feature admin* group, and a *map user* group. The data accessibility of these groups will be as follows:

- **metadata admin** group–read and write permissions on metadata; read permission on all other data
- **feature admin** group–read and write permissions on feature data; read permission on all other data
- **map user** group–read permission on all data

Symbol lists that manage visibility of data also need to be established. Each user's symbol list includes some set of GemStone dictionaries through which the GemStone Smalltalk compiler will search to resolve object references in an application program. A symbol list is a part of each user's profile that additionally contains a user name, password, and default segment. The combination of user groups, symbol lists, and user profiles provides a complete set of mechanisms to control the accessibility and visibility of data in the GemStone database.

5.0 CONCLUSION

Our experience with ObjectStore was very favorable. In contrast to the relative ease with which we integrated support for ObjectStore into OVPF, however, GemStone presented much greater difficulty. Because two copies of objects are maintained, one at the server and one at the client, GemStone actually maintains a mirror of the application program in the server. This allows the objects' behavior—not just their state—to become part of the database. A consequence is that many programming techniques that are suitable in a single-user, memory-based programming environment do not work in a database programming environment. We found that fundamental changes in some of OVPF's lowest-level data structures will have to be made before we can complete the migration of VPF data to the GemStone database. Only the metadata has been migrated to date.

Both ODBMSs provide acceptable storage of and access to OO data through their Smalltalk interface. In summary, ObjectStore appears less time consuming to integrate due to its relatively simpler functionality. GemStone, on the other hand, provides more sophisticated functionality, but with greater effort at integration. Another consideration is that ObjectStore provides more flexibility and control in the physical database design with the use of segments for clustering of data, resulting in potentially better access time. Finally, GemStone has built-in access control methods for regulating the visibility and accessibility of data to various users that ObjectStore does not currently provide (although Object Design, Inc. plans to include these with its next revision).

Another factor in the selection of an ODBMS for continued development work has been considered for the last several months. In anticipation of future DMA directions in software development, we have been preliminarily investigating the feasibility of transitioning OVPF development to Ada 95. At this time, Object Design, Inc. has informed us that they expect a greater than 50% probability that an Ada 95 interface to ObjectStore will be developed within the next year or so. GemStone Systems, Inc., however, has no plans at this time to support Ada 95 for GemStone. We believe, based on availability of software development support tools and databases, that for the near future, Smalltalk should continue to be our development environment. We are prepared, however, to transition to Ada 95 when commercial support for it has been significantly established.

6.0 ACKNOWLEDGMENTS

We wish to thank our sponsor, DMA, Mr. Jim Kraus and Mr. Jake Garrison, program managers, for sponsoring this research. We would also like to thank Mr. Mike Harris for performing the technical review of this report.

7.0 REFERENCES

Arctur, D. K., J. F. Alexander, M. A. Cobb, M. J. Chung, and K. B. Shaw, "OVPF Report: Evaluation of Illustra Hybrid Object-Relational DBMS," Naval Research Laboratory, Stennis Space Center, MS, 1995a.

Arctur, D. K., J. F. Alexander, M. A. Cobb, M. J. Chung, and K. B. Shaw, "OVPF Report: Issues and Approaches for Spatial Topology in GIS," NRL/MR/7441--96-7719, Naval Research Laboratory, Stennis Space Center, MS, 1995b.

Chung, M. J., M. A. Cobb, K. B. Shaw, D. K. Arctur, and J. F. Alexander, "OVPF Report: Network Investigation Results," Naval Research Laboratory, Stennis Space Center, MS, 1995a.

Chung, M. J., M. A. Cobb, K. B. Shaw, and D. K. Arctur, "An Object-Oriented Approach for Handling Topology in VPF Products," *Proc. GIS/LIS '95*, Nashville, TN, Vol. 1, pp. 163–174, 1995b.

Cobb, M. A., M. J. Chung, K. B. Shaw, and D. K. Arctur, "A Self-Adjusting Indexing Structure for Spatial Data," *Proc. GIS/LIS '95*, Nashville, TN, Vol. 1, pp. 182–192, 1995.

DeWitt, D. J., D. Maier, P. Futtersack, and F. Velez, "A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database System," in *Proceedings 16th International Conference on Very Large Data Bases*, Brisbane, Australia, pp. 107–121, 1990.

"Military Standard: Vector Product Format," Draft Document No. MIL-STD-2407, Defense Mapping Agency, Fairfax, VA, 1993.

Nelson, R. C. and H. Samet, "A Population Analysis for Hierarchical Data Structures," in *Proceedings of the SIGMOD Conference*, pp. 270–277, San Francisco, CA, 1987.

Shaw, K. B., M. J. Chung, M. A. Cobb, D. K. Arctur, J. F. Alexander, and E. Anwar, "Object-Oriented Database Exploitation Within the GGIS Data Warehouse: Initial Report," NRL/FR/7441--95-9639, Naval Research Laboratory, Stennis Space Center, MS, 1995.

Stonebraker, M., T. Sellis, and E. Hanson, "An Analysis of Rule Indexing Implementations in Data Base Systems," in *Proceedings of the First International Conference on Expert Database Systems*, Charleston, SC, pp. 353–364, April 1986.